Unit –4: Inheritance

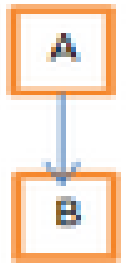
Introduction to Inheritance and Reusability

- It is the process the by which object of one class derived (acquired) the properties of object of another class.
- The mechanism of deriving a new class from an old class is called as inheritance.
- In inheritance, the old class is referred as the **base class** and the new class is called as the **derived class or subclass.**
- The derived class inherits some or all the members from the base class.
- A class can also inherit properties from more than one class or from more than one level.
- A derived class with only one base class is called as single inheritance.

- A derived class with several base classes is called multiple inheritances.
- The traits of one class may be inherited by more than one class is called as hierarchical inheritance.
- The mechanism of deriving a class from another ‘derived class’ is called as multilevel inheritance.

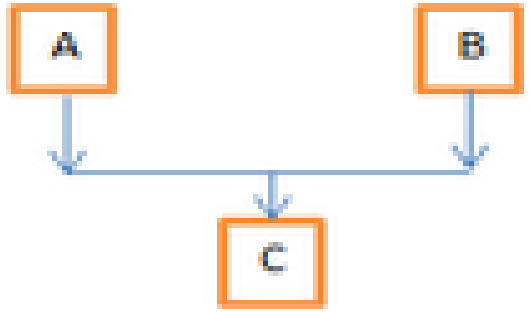
Types of Inheritance:

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid Inheritance

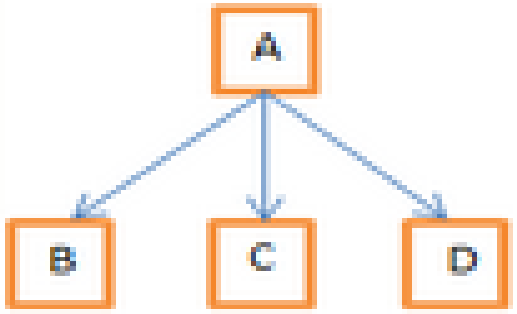


Single Inheritance

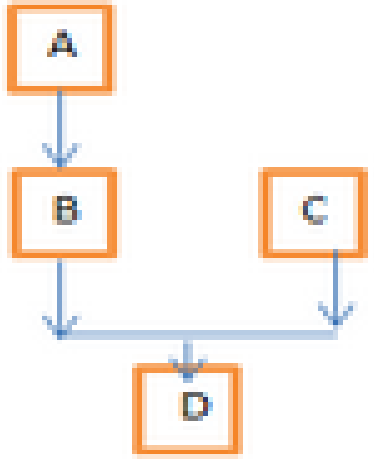
Multilevel Inheritance



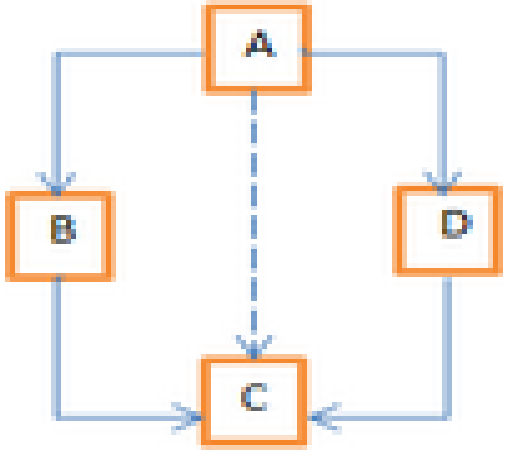
Multiple Inheritance



Hierarchical Inheritance



Hybrid Inheritance



Multipath Inheritance

Simple Inheritance: Using public, Private and protected derivation

- A derived class can be defined by specifying its relationship with the base class.

Syntax:

```
class derived_class_name: visibility_mode base_class_name
{
        . . . . . .
}
```

- The colon indicates that the derived-class-name is derived from the base-class-name.
- The visibility mode is optional. It may be either public or private.
- The default visibility mode is private.
- Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:

Exa-1

```
class B : public A   // public derivation
{
        Members of B
}
```

Exa-2

```
class B : private A  // private derivation
{
        Members of B
}
```

Exa-3

```
class B : A   // private derivation
{
        Members of B
}
```

- **Public inheritance** makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.
- **Protected inheritance** makes the public and protected members of the base class protected in the derived class.
- **Private inheritance** makes the public and protected members of the base class private in the derived class.

➢Note: private members of the base class are inaccessible to the derived class.

```
class Base {
        public:
        int x;
        protected:
        int y;
        private:
        int z;
};
class A: public Base {
  // x is public
  // y is protected
  // z is not accessible from A
};
```

```
class B: protected Base
{
  // x is protected
  // y is protected
  // z is not accessible from B
};

class PrivateDerived: private Base
{
  // x is private
  // y is private
  // z is not accessible from C
};
```

	Derived Class	Derived Class	Derived Class
Base Class	**Private Mode**	**Protected Mode**	**Public Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Accessibility in public Inheritance

Accessibility	Private members	Protected members	Public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Accessibility in protected Inheritance

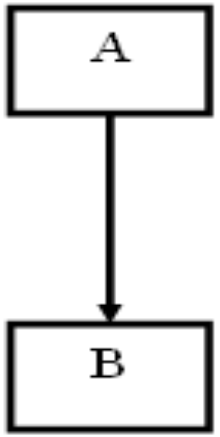
Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

Accessibility in private Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

Single Inheritance

- In single level inheritance there is only one base class and one derived class.
- Figure of single level inheritance given below:



- In above figure, class A is called as Base Class (Old Class) and class B is called as Derived class (New Class).
- The derived class B derived all or some properties (members) of the base class A.
- The derived class B has its own properties as well as derived properties of base class A.
- Using the object of derived class B we can call the members of Derived class as well as members of base class A which are derived inside the derived class.

```
class circle
{
        protected:
        float a;
        int r;
        public:
        void getdata()
        {
                cout<<"Enter r=";
                cin>>r;
        }
        void area()
        {
        a=3.14*r*r;
        cout<<"Area="<<a;
        }
};
```

```
class circle_ext:public circle
{
        protected:
        float p;
        public:
        void peri( )
        {
                p=2*3.14*r;
                cout<<"Peripheral="<<p;
        }
};
```

```
void main()
{
        clrscr( );
        circle c1; //base class
        c1.getdata( );
        c1.area( );
        circle_ext c2; //derived class
        c2.getdata( );
        c2.area( );
        c2.peri( );
        getch();
}
```

Output:

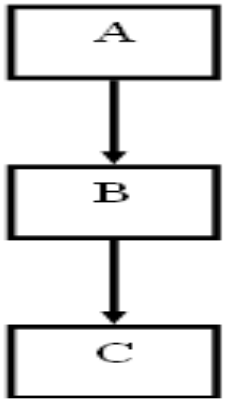
```
Enter r=2
Area=12.56
Enter r=3
Area=28.26
Peripheral=18.84
```

```
class A
{
    int a = 4;
    int b = 5;
    public:
            int mul()
            {
                    int c = a*b;
                    return c;
            }
};
class B : private A
{
    public:
    void display()
    {
        int result = mul();
        cout <<result;
    } };
```

```
int main()
{
   B b;
   b.display();
b.mul();

    return 0;
}
```

Multilevel Inheritance

- When a class is derived from another derived class i.e., derived class act as a base class, such type of inheritance is known as multilevel inheritance.
- In multilevel inheritance, the class is derived from another derived class.



- Using the object of derived class B we can call the members of derived class B as well as members of base class A which are derived inside the derived class
- Using the object of derived class C we can call the members of derived class C as well as members of base class A and B which are derived inside the derived class.

```
class A
{
        public:
         int a;
        void get_A_data()
        {
                cout << "Enter value of a: ";
                cin >> a;
         }
};
class B : public A {
        public:
        int b;
        void get_B_data()
        {
                cout << "Enter value of b: ";
                cin >> b;
        }
};
```

```
class C : public B
{
        private:
        int c;
        public:
        void get_C_data()
        {
        cout << "Enter value of c: ";
        cin >> c;
        }
        void sum()
        {
                int ans = a + b + c;
                cout << "sum: " << ans;
         }
};
```

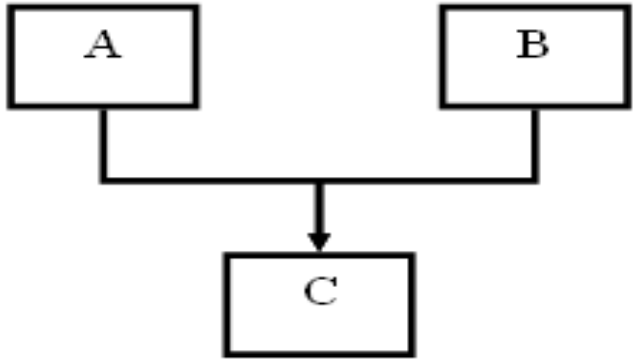
```
int main()
{
        C obj;

         obj.get_A_data();
         obj.get_B_data();
        obj.get_C_data();
        obj.sum();
        return 0;
}
```

Output:

```
Enter value of a: 4
Enter value of b: 5
Enter value of c: 9
sum: 18
```

Multiple Inheritance

- When two or more base classes are used in derivation of a class, it is called as multiple inheritances.
- In multiple inheritances, one derived class is derived from more than one base class.



- In above figure, class A and class B is called as Base Class (Old Class) for class C. And class C is called as Derived class (New Class) for class A and class B.
- Using the object of derived class C we can call the members of derived class C as well as members of base class A and base class B which are derived inside the derived class

```
class add
{
    public:
        int x = 20, y = 30;
        void sum()
        {
            cout <<x+y << endl;
        }
};
 class Mul
{
    public:
        int a = 20;
        int b = 30;
        void mul()
        {
            cout <<a*b << endl;
        }
 };
```

```
class Sub
{
    public:
        int a = 50, b = 30;
        void sub()
        {
            cout <<a-b << endl;
        }
};
class Div
{
     public:
        int a = 150, b = 30;
        void div()
        {
            cout <<a/b << endl;
        } };
```

```
class derived: public add, public Div, public
Sub, public Mul
{
public:
        int p = 12;
        int q = 5;
        void mod()
        {
            cout << p % q << endl;
        }
};
```

```
int main ()
{
    derived dr;
    dr.mod();
    dr.sum();
    dr.mul();
    dr.div();
    dr.sub();
}
```

Ambiguity in Multiple Inheritance

- The most obvious problem with multiple inheritance occurs during function overriding.
- Suppose, two base classes have a same function which is not overridden in derived class.
- If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

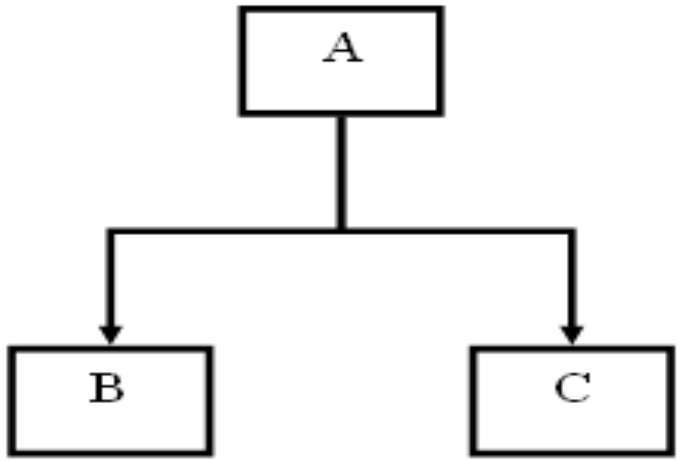
Ambiguity in Multiple Inheritance

- The most obvious problem with multiple inheritance occurs during function overriding.
- Suppose, two base classes have a same function which is not overridden in derived class.
- If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

```
class base1 {
  public:
     void someFunction( ) {....}
};
class base2 {
    void someFunction( ) {....}
};
class derived : public base1, public base2
{
};

int main() {
    derived obj;
    obj.someFunction() // Error!
}
```

Hierarchical Inheritance

- When a single base class is used for derivation of two or more classes it is known as hierarchical inheritance.
- In hierarchical inheritance, more than one derived class is derived from a single base class.



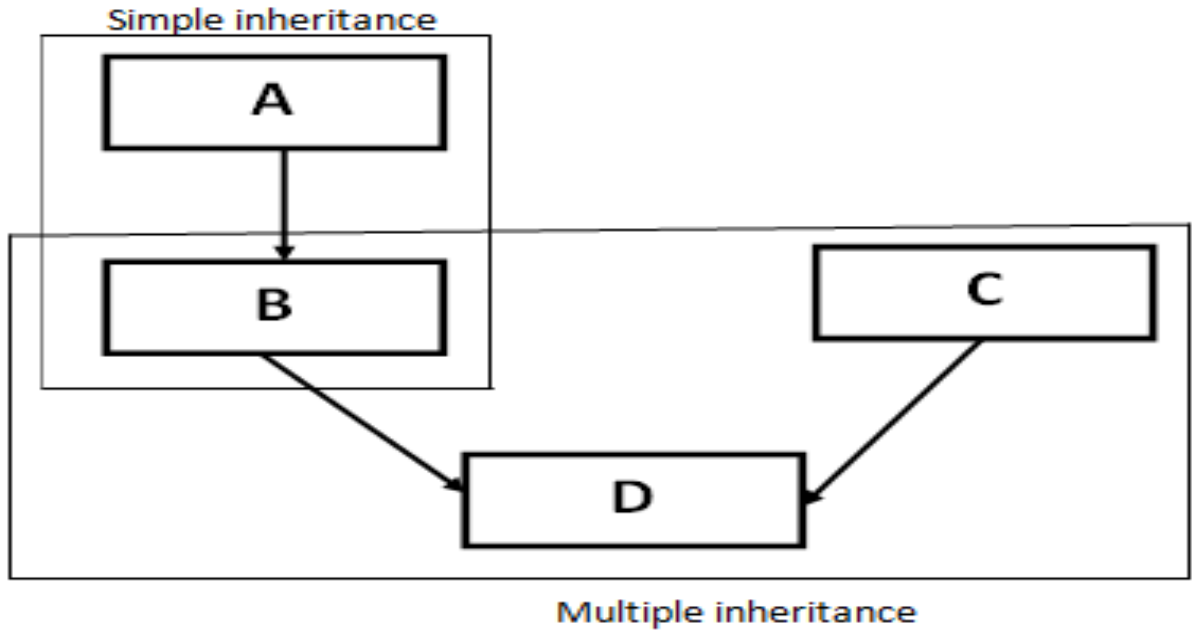
- Using the object of derived class B we can call the members of derived class B as well as members of base class A which are derived inside the derived class.
- Using the object of derived class C we can call the members of derived class C as well as members of base class A which are derived inside the derived class.

```
class A
{
        public:
        int x, y;
        voidgetdata()
        {
        cout<< "Enter value of x and y:\n";
        cin>> x >> y;
        }
};
class B : public A
{
        public:
        void product()
        {
        cout<< x * y <<endl;
        }
};
```

```
class C : public A
{
        public:
        void sum()
        {
        cout<< x + y;
        }
};
int main()
{
    B obj1;
    C obj2;
obj1.getdata();
obj1.product();
obj2.getdata();
obj2.sum();
}
```

Hybrid Inheritance

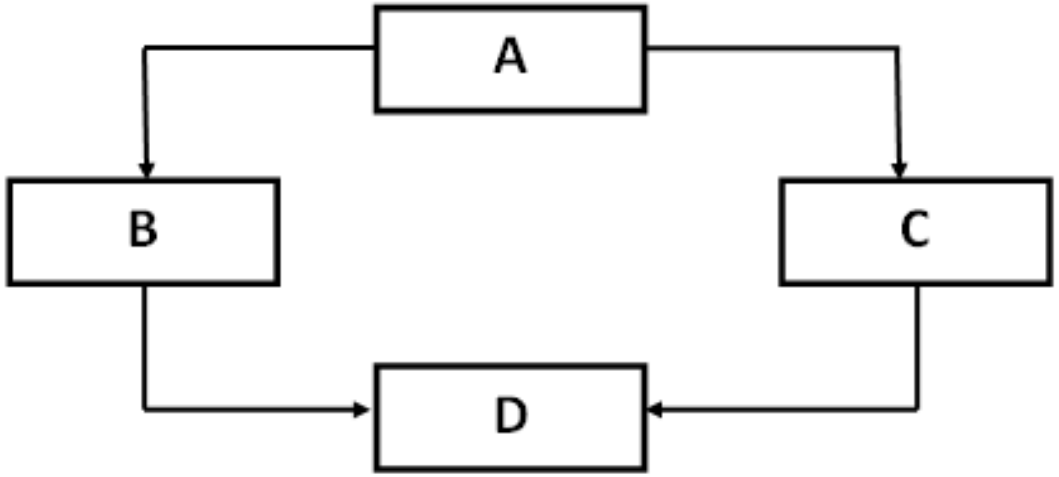
- The combination of more than one type of inheritance is known as hybrid inheritance.
- Figure of hybrid inheritance given below:



- In above diagram class B is derived from class A which is single inheritance
- and then Class D is inherited from B and class C which is multiple inheritance.
- So single inheritance and multiple inheritance jointly results in **hybrid inheritance**.

Multipath Inheritance

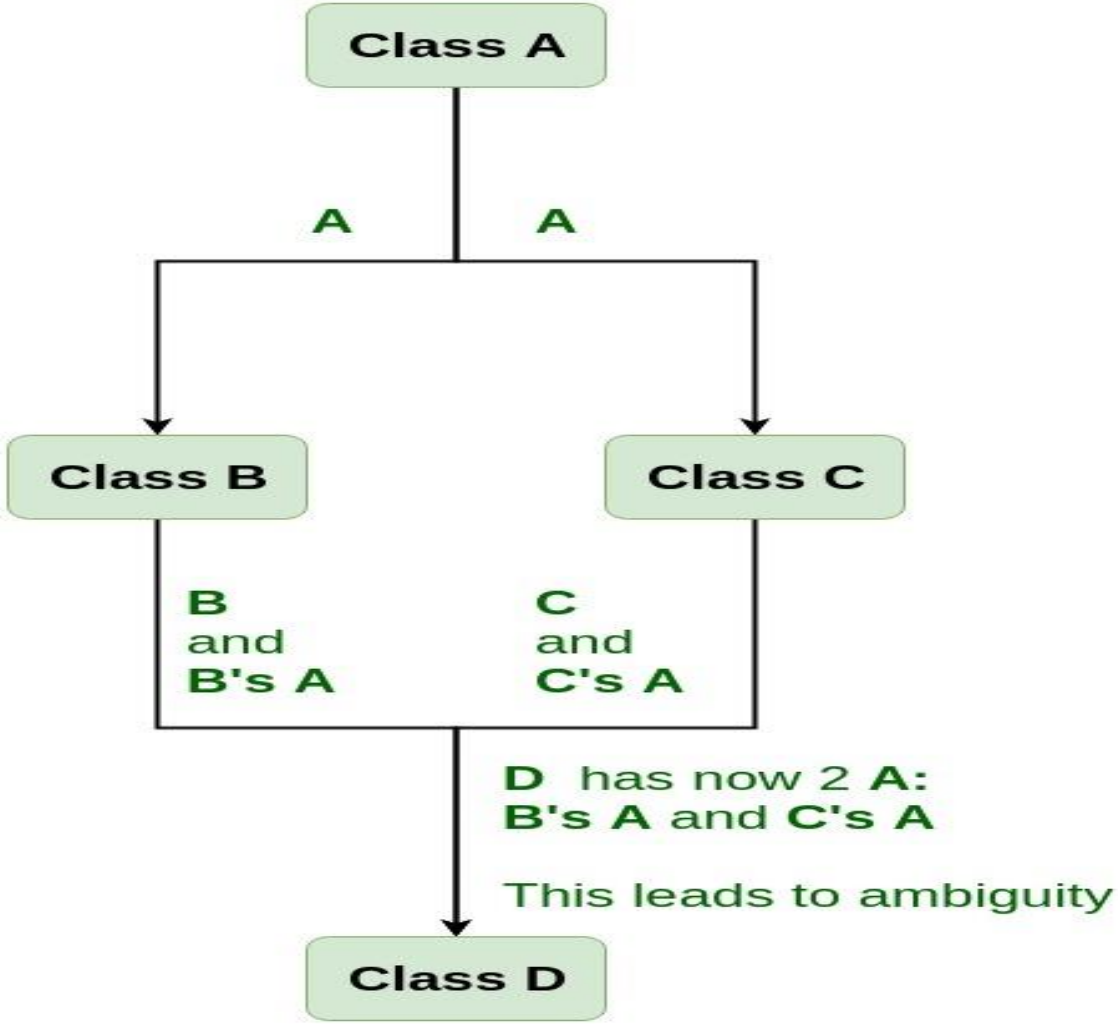
- It refers to the process of deriving a class from two or more classes which is being derived from the same base class.
- This type of inheritance involves other inheritance like multiple, multilevel, hierarchical etc.



- In the above figure, there is hierarchical inheritance between class A, class B and class C. And multiple inheritance exists between class B, class C and class D.

Virtual Base Class

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.
- Such a base class is known as virtual base class.
- This can be achieved by preceding the base class’ name with the word virtual.
- virtual can be written before or after the public



- To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as
- Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class.

There are a few details that one needs to remember.

- Virtual base classes are always created before non-virtual base classes. This ensures all bases are created before their derived classes.
- Note that classes B and C still have calls to class A, but they are simply ignored when creating an object of class D. If we are creating an object of class B or C, then the constructor of A will be called.
- If a class inherits one or more classes with virtual parents, the most derived class is responsible for constructing the virtual base class. Here, class D is responsible for creating class A object.

```
class A
{
        public:
        int i;
        void display()
        {
                cout<<i;
        }
};
class B : virtual public A
{
        public:
        int j;
};
class C: virtual public A
{
        public:
        int k;
};
```

```
class D: public B, public C
{
        public:
        int sum;
};
void main()
{
D ob;
ob.i = 10; //only one copy of i is inherited.
ob.j = 20;
ob.k = 30;
//ob.B::i;
//ob.C::i;
ob.sum = ob.i + ob.j + ob.k;
cout << “Sum is : ”<< ob.sum <<”\n”;
}
```

Constructors in base and derive classes

- Base class constructors are always called in the derived class constructors.
- Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.
- Whether derived class's default constructor is called or parameterized is called, base class's default constructor is always called inside them.
- To call base class's parameterized constructor inside derived class's parameterized constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Order of constructor and Destructor call for a given order of Inheritance

Order of Inheritance

Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

```
class Parent
{
    public:
     Parent()
    {
        cout << "Inside base class" << endl;
    }
};

class Child : public Parent
{
    public:
     Child()
    {
        cout << "Inside sub class" << endl;
    }
};
```

```
int main()
{
       Child obj;
      return 0;
}
```

Output:

Inside base class
Inside sub class

```
class Parent1
{
    public:
    Parent1()
    {
    cout << "Inside first base class" << endl;
    }
};
class Parent2
{
    public:
    Parent2()
    {
            cout << "Inside second base class"
    }
};
```

```
lass Child : public Parent1, public Parent2
{
    public:
      Child()
    {
        cout << "Inside child class" << endl;
    }
};
int main()
{
      Child obj1;
      return 0;
}
```

Important Points:

- Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
- To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.
- The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

```
class base
{
        protected:
        int a;
        public:
        base()
        {
                a=0;
        }
        base(int a1)
        {
                a=a1;
        }
        void display()
        {
        cout<<"A="<<a;
        }
};
```

```
class derived:public base
{
        protected:
        int b;
        public:
        derived() {
                b=0;
        }
        derived(int a1,int b1):base(a1)
        {
                b=b1;
        }
        void display()
        {
        cout<<"A="<<a<<"B="<<b<<endl;
        }
};
```

```
void main( )
{
        clrscr();
        base a1(10);
        a1.display();
        derived b1(20,30);
        b1.display();
}
```

Output:

Abstract Class

- Sometimes implementation of all functions cannot be provided in a base class because we don’t know the implementation. Such a class is called an abstract class.
- Abstract Class is a class which contains at least one Pure Virtual function in it.
- Abstract classes are used to provide an Interface for its sub classes.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

C++ Pure Virtual Functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named calculateArea() in the Shape. Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the calculateArea() function with implementation.

A pure virtual function doesn't have the function body and it must end with = 0

```
class Shape
{
    protected:
    float dimension;
    public:
    void getDimension()
   {
        cin >> dimension;
    }
            virtual float calculateArea() = 0;
};

class Square : public Shape {
   public:
    float calculateArea() {
        return dimension * dimension;
    }
};
```

```
class Circle : public Shape
{
   public:
    float calculateArea()
   {
        return 3.14 * dimension * dimension;
    }
};
```

```
int main()
{
    Square square;
    Circle circle;
   cout << "Enter the length of the square: ";
    square.getDimension();
    cout << square.calculateArea() << endl;

    cout << "\nEnter radius of the circle: ";
    circle.getDimension();
    cout << circle.calculateArea() << endl;

    return 0;
}
```

Output:

```
Enter the length of the square: 4
Area of square: 16
Enter radius of the circle: 5
Area of circle: 78.5
```

advantages and disadvantages of inheritance

Advantages of Inheritance:

- The main advantage of the inheritance is that it helps in reusability of the code. The codes are defined only once and can be used multiple times.
- Through inheritance a lot of time and efforts are being saved.
- It improves code readability as you don’t have to rewrite the same code repeatedly
- The program structure is short and concise which is more reliable.
- The codes are easy to debug. Inheritance allows the program to capture the bugs easily
- Inheritance makes the application code more flexible to change.

Disadvantages of Inheritance:

- The main disadvantage of the inheritance is that the two classes(base class and super class) are tightly coupled that is the classes are dependent on each other.
- If the functionality of the base class is changed then the changes have to be done on the child classes also.
- If the methods in the super class are deleted then it is very difficult to maintain the functionality of the child class which has implemented the super class’s method.
- It increases the time and efforts take to jump through different levels of the inheritance.